Proceedings of the UK e-Science All Hands Meeting 2006, © NeSC 2006, ISBN 0-9553988-0-0

Preserving Scientific Data with *XMLArch* *

Peter Buneman James Cheney **Carwyn Edwards** Irini Fundulaki

{opb,jcheney,cedward1,efountou}@inf.ed.ac.uk

Abstract

Scientific databases are continuously updated either by adding new data or by deleting and/or modifying existing data. It is fairly obvious that it is crucial to preserve historic versions of these databases so that researchers can access previous findings to verify results or compare old data against new data and new theories. In this poster we will present XMLArch that is used in archiving scientific data stored in relational or XML databases. XMLArch builds upon and extends previous archiving techniques for hierarchical scientific data.

1 Introduction

The Web is now the most important means of publishing scientific information. In particular, an increasing number of scientific databases are published on the Web, allowing scientists to exchange their research results. Most of these databases are continuously updated either by adding new data or by deleting and/or modifying existing data. In order to preserve the scientific record it is crucial to keep versions of these databases. Researchers should be able to validate previous findings and to compare old data against new data and new theories. It is reported in [3] that archiving is a ubiquitous problem. Even databases that record ostensibly fixed data, such as the results of experiments or simulations have associated metadata, such as classification information and annotation, and this metadata is almost always subject to change.

In this paper we report on progress in constructing *XMLArch*, a generic system for archiving scientific data represented in XML. Based on this we have constructed a simple tool that one can simply point at a relational database. It extracts the data into a default XML format with associated key information and then incorporates the XML into an incremental archive. Not only does this preserve successive versions of the database, it preserves them in a fashion that is independent of any specific relational database management system and makes possible temporal queries on data.

We use the IUPHAR pharmaceutical database [8] as a concrete example to discuss the problems arising with archiving real scientific data. IUPHAR (International Union of Pharmacology) was founded in 1959 and one of its main objectives is to foster international cooperation in pharmacology by promoting cooperation between societies that represent pharmacology and related disciplines throughout the world.

The IUPHAR database is continuously updated and it is published two to three times a year. Archiving the different versions of the database is crucial since it is widely cited by researchers in their work who need to access its different versions. The IUPHAR curators keep versions of the database as database dumps (complete snapshots) and only the latest version of the database is live. Such snapshots unfortunately cannot easily be queried and it is extremely cumbersome for one to access historic information, for example the change history for a given receptor. It is also evident that there is a significant storage overhead in keeping all the versions of the database.

Apart from this brute-force approach in archiving scientific data, other approaches based on storing the differences (or deltas) between the different versions of text documents are also common. These approaches, that are based on line-diff algorithms, clearly conserve space and scale well. However, retrieving an older version might involve either undoing or applying many deltas. In a similar manner, finding how an element has evolved is also a problem and may require complicated reasoning using the recorded deltas. This is because the current approaches based on differences do not preserve the structure of the database, hence the identity of objects is lost.

In this poster we will demonstrate *XMLArch* that is used in archiving scientific data stored in relational or XML databases. The approach has been presented in [2] and is based on the idea of merging all the database versions into a single archive. It takes advantage of the hierarchi-

* This project has been supported in the Digital Curation Centre, which is funded in part by the EPSRC eScience core programme.

cal structure of the data and leverages the strong key structures which are common in the design of scientific datasets. We will also present the benefits of combining archiving as done in *XMLArch* and XML compression, using for the latter the state of the art techniques as surveyed in [5].

2 Archiving Scientific Data

In this section we give a high level overview of the *XMLArch* archiver. As already mentioned, we choose the archiving approach presented in [2]. This work dealt with archiving scientific data that is stored in some hierarchical data format (such as XML). In the case of relational data we first need to extract the relational data into a hierarchical format, in this case to XML. To do this we built a simple relational database extraction tool, the output of which is then passed into the archiving tool itself and optionally on to a compressor.

The idea for the initial relational database extractor was that it should not need to know anything at all about the semantics of the domain data. To facilitate this the extractor uses a simple schema-agnostic XML format to represent the extracted relational data, making the implementation fairly portable across different databases and domains. The only input required is access to the relational database itself. The motivation for this simplistic approach being that often the person archiving a dataset has little or no prior experience with the data (e.g. a Systems Administrator).

This *point and extract* behavior makes the extractor behave very much like conventional relational database backup tools. The eventual aim is that as much as possible of the relational information in the database will be preserved along with the XML snapshot. The theory being that as long as the data is preserved along with some form of structural description, possibly no more than a textual description of the relations, then someone in the future will be able to reconstruct the relations around the data. Obviously if we can store as much of the schema as possible in a standard form this reduces the work needed to recreate it.

One last aim was that the data extractor should not attempt to do too much. Tools already exist from virtually every relational database vendor to extract data into XML. The extractor component of this project should be seen as a simplest possible tool to use in order to get the data out of the database in the absence of other more suitable options.

Once the relational data is extracted it is archived by the XML archiver submodule of *XMLArch* using an approach based on the one introduced in [2]. The archiving techniques presented in that work stem from the requirements and properties of scientific databases: first of all, much scientific data is kept either in well-organized hierarchical data formats, or it has an inherent hierarchical structure; second, this hierarchically structured data usually has a key structure which is explicit in the design. The key structure provides a canonical identification for every part of the document, and it is exactly this structure that is the basis of the technique in [2].

In [2] the idea behind archiving is based on:

1. *identifying the correspondence and changes between two given versions based on keys* and
2. *merging the different versions using these keys in one archive*.

Identifying correspondences between versions using keys is different from the *diff-based* ([7]) approaches used in most of the existing tools. *Diff-based* approaches are based on minimum edit distances between raw text lines and use no information about the structure of the underlying data. Our key based approach on the other hand attempts to unify objects in different data versions based on their keyed identity. In this way, the archive can not only preserve the semantic continuity between the elements but also efficiently support queries on the history of those elements.

The merging of different versions into one archive is again different from the *diff-based* approaches that store the deltas from version to version independently. Occurrences of elements are identified using the key structure and stored only once in the final archive. A sequence of numbers (*timestamp*) is used to record the sequence of versions in which an element appears. This timestamp is conceptually stored with every element in the hierarchy. Taking advantage of the hierarchical nature of the data, the timestamp in an element is stored only when it is different from that of its parent. In this way, a fair amount of space overhead is avoided by inheriting timestamps.

The final archive is stored as an XML document. XML was chosen as the archive format for a number of reasons: for one thing the hierarchical models used in many biological databases are very similar to the XML data model. In addition, XML is currently the most prevalent textual format for hierarchical data. Because of this there are a significant number of commercial and open source tools that are readily available to manipulate XML. Given that one of the most important considerations when archiving is to make sure that the data is retrievable at a later date, using a widespread, *human readable* text-based format would seem to increase the likelihood that the data will be accessible in the future.

The final XML archive is then optionally passed into standard text or XML compression tools to further reduce the storage requirements of the archive. As the same principles for archiving as noted in [2] are used by the XML archiver, the same benefits in terms of storage overhead reductions noted in that work apply. As was shown in [2] the resulting output is often particularly well suited to XML specific compressors such as XMill [9]. This is unsurprising given the hierarchical models underlying the source databases and is supported by the findings of [4].

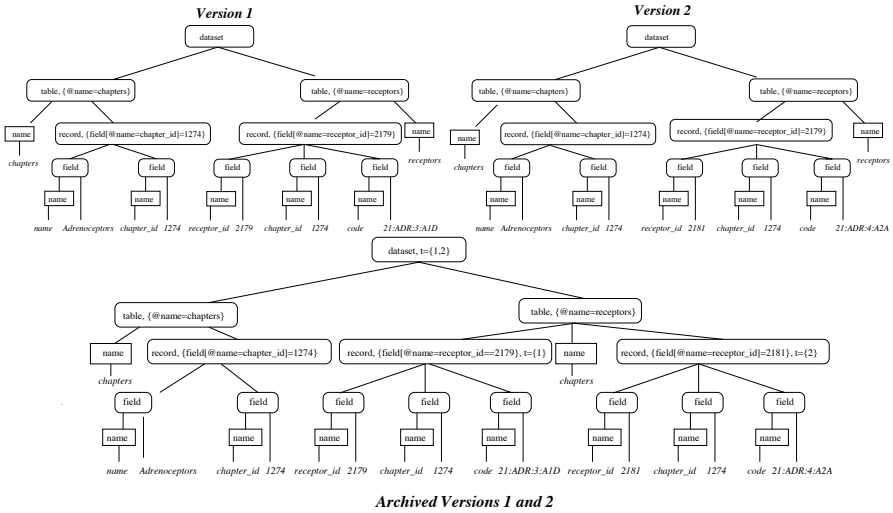

Figure 1. Archived Versions of the IUPHAR database

3 An Example: Archiving the IUPHAR database

In this section we present an example to demonstrate how *XMLArch* is used to archive IUPHAR relational data. To present our approach we use a simple example of IUPHAR data.

We consider tables chapters and receptors shown below (the primary keys for all tables are underlined). The first stores the receptor families where each family has a $chapter\_id$ (primary key) and a $name$. Table receptors stores the $name$ and $code$ of receptors. Attribute $receptor\_id$ is the primary key for the table and attribute $chapter\_id$ indicates the receptor family to which the receptor belongs to.

Source relational schema R:

chapters($\underline{chapter\_id}$, $name$)
receptors($\underline{receptor\_id}$, $chapter\_id$, $name$, $code$)

The XML DTD to which this data is published is shown below:

```
1. <!ELEMENT dataset (table*)>
2. <!ELEMENT table (record*)>
3. <!ATTLIST table name #PCDATA>
4. <!ELEMENT record (field+)>
5. <!ELEMENT field #PCDATA>
6. <!ATTLIST field name #CDATA
7.           keyfield (true|false) 'false'>
```

Element `table` stores information about a relational table. Attribute `name` records the name of the table (line 3). Each `table` element has one or more `record` elements (line 2) where such an element is defined for each tuple of the corresponding relational table. A `record` element has one or more `field` subelements (line 4). A `field` element is defined for an attribute of the relational table with XML attribute `name` to record the name of the relational attribute (line 6). Attribute `keyfield` records whether the attribute participates in the primary key of the relational table or not (line 7).

In addition to this XML data, we also need a way to identify the *XML elements* in the extracted XML document. This is done by using the notion of *XML keys* introduced in [1]. In this work an XML element is *uniquely identified* by the values of a subset of its *descendant* elements.

For example, in our case a `table` element is uniquely identified by its `name` attribute. A `record` element within the `table` element defined for the relational table chapters (i.e., the `table` element with value *chapters* for its `name` attribute) is uniquely identified by the value of its `field` subelement that corresponds to the relational attribute $chapter\_id$. In a similar manner, a `record` element within a `table` element that corresponds to the receptors relational table is uniquely identified by the value of its `field` subelement defined for the $receptor\_id$ relational attribute. This key information is defined by means of XPath [6] expressions as advocated in [1]. In *XMLArch* the annotator component, a sub component of the archiver, records with each keyed element its XML key and value whenever applicable (e.g. elements `table` and `record`). These key annotations are used during the merging of a new version of the database with the archived version to unify element identities.

We show in the upper part of Figure 1 two versions of the database. The difference between Version 1 and Version 2 is that receptor with receptor_id equal to 2179 has been deleted and receptor with receptor_id equal to 2181 has been added.

The archived XML document is also shown in Figure 1. One can observe that elements that exist in both versions are stored only once (e.g., the `table` elements and the `dataset` element). Notice that there is a timestamp ($t = \{1,2\}$) associated with the `dataset` element that indicates that this element (as well as all its descendants that do not have a timestamp) are present in both versions. Observe that the `record`

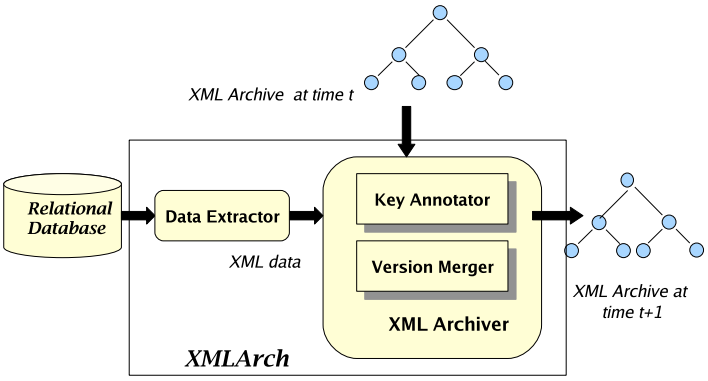

Figure 2. *XMLArch* **Architecture**

element deleted in the second version has a timestamp equal to "1" ($t = \{1\}$) whereas the `record` element added in the second version has timestamp equal to "2" ($t=\{2\}$).

4 System Architecture

The architecture of the system is shown in Figure 2. The **Data Extractor** is responsible for extracting the relational data into the XML format discussed previously. This component reads the schema of the database (tables and constraints such as primary keys) and the instances (i.e., tuples) and exports the database in XML.

The exported data is then passed to the **XML Archiver** that is responsible for creating the XML archive. This module consists of the **Key Annotator** and **Version Merger** submodules. The former is responsible for annotating each XML element with its key, and the latter for merging the latest XML archive (archive at time `t`) with the new version to produce the archive at time `t+1`.

References

[1] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW*, 2001.

[2] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving Scientific Data. *TODS*, 2004.

[3] The WWW Virtual Library of Cell Biology. http://vlib.org/Science/Cell_Biology/databases.shtml.

[4] J. Cheney. Compressing XML with Multiplexed Hierarchical Models. In *In Proc. IEEE Data Compression Conference (DCC)*, 2001.

[5] J. Cheney. An Empirical Evaluation of Simple DTD-Conscious Compression Techniques. In *WebDB*, 2005.

[6] J. Clark and Steve DeRose. XML Path Language (XPath) 1.0. W3C Recommendation, 1999. http://www.w3c.org/TR/xpath.

[7] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR #41, Bell Telephone Laboratories, 1976.

[8] IUPHAR. Receptor Database. `http://www.iuphar-db.org`.

[9] H. Liefke and D. Suciu. XMill: an Efficient Compressor for XML Data. In *SIGMOD*, pages 153–164, 2000.